Using Massive Atomic Write Operations in GPU Applications

Abstract

Many parallel applications involve simultaneous write updates to shared memory objects. To ensure program correctness, developers need to impose an ordering among different shared memory write updates. One design choice is to use private data objects for individual threads and subsequently apply sophisticated algorithms to merge write results with implicit barriers. Another design choice is to simply use hardware atomics so that write updates to shared data objects are guaranteed to happen in a particular order without introducing data races. The first method has been a widely adopted choice, since atomic write operations are more expensive than non-atomic write updates. Moreover, native atomic writes may serialize computation tasks when multiple threads update the same memory location within a short time period. Therefore, native atomic writes are mostly used in scenarios such as synchronization and concurrency control. However, due to the explosive growth of parallelism in both software (threads) and hardware (cores), designing applications with absolutely no write collisions and only implicit synchronization barriers has become an extremely challenging task in terms of the complexity of algorithms and the growing memory space requirement for private data objects. We are facing programming challenges and we need to address several open questions. The first one is whether we should consider applying more native atomic writes for shared memory data updates in order to alleviate the algorithm complexity and memory requirement problem. Is there any potential for improving program efficiency with smartly scheduled native atomic write updates? If so, how many atomic writes should be allowed, to ensure both efficiency and coding productivity? In this work, we conducted a systematic exploration to address these questions. Despite the traditional perception that we should avoid large numbers of atomic writes, using massive atomic writes and scheduling them appropriately is in fact a simple and yet effective approach for programming many-core GPUs. The key is to detect collisions, scatter the conflicted writes into different time intervals, and reduce the computation serialization. We propose two basic schemes in this paper: one is parallel reduction based and the other is data/computation re-scattering based. We have found that not only irregular parallel applications that are hard to implement without explicit synchronization or atomic operations can benefit from these techniques, but also traditional regular applications such as sparse matrix vector multiplications can be improved significantly using these techniques. We have implemented fast and lightweight parallel collision detection and reduction techniques, and achieved up to a 1.69 times speedup for a set of representative scientific computation kernels.

1. Introduction

General Purpose Graphic Processing Units (GPGPUs) have become increasingly popular in different types of systems from mobile devices to supercomputing clusters in recent years. More and more developers are porting applications in a variety of domains from CPUs to GPUs in order to benefit from the tremendous computing throughput and cost/power efficiency of GPUs. In the porting of parallel CPU applications to GPUs, there are three major obstacles that arise due to the limits of data parallel architectures. These are control flow divergence, irregular memory reference and memory atomic conflicts. The first two have been extensively studied from the aspects of both software and hardware optimization [17] [7] [4]. The memory atomic operation, which is a fundamental component for an wide domain of applications with data parallelism including the Recognition, Mining, and Synthesis (RMS) application domain, is a double-edge sword. On one hand, it can help the communication between different threads. On the other hand, it many cause resource contention and thread serialization because it is an exclusive operation with respect to every individual thread. The problem has primarily been investigated from the hardware aspect [9]. Very few applications utilizing intensive atomic operations have been ported to the GPU. A few recent efforts include but are not limited to clustering [10], hashing [1]), image processing (histogramming [11]), and dynamic bio-simulation [2]. The reason is that a GPU chip can easily accomodate hundreds of cores and tens of thousands of threads, on which write contention caused by atomic updates to conflicting memory locations immediately becomes intractable compared to that on a typical Chip Multiprocessor with a few cores.

Atomic operations are expensive compared to regular memory operations. As a result, in some studies, people try to avoid them by designing atomics-free algorithms using thread-private memory objects [6]. However, these types of algorithms are dependent on the size of memory and their efficiency is therefore limited. Due to the importance of atomic operations, architectural support has been provided since very early generation GPUs. For example, NVIDIA started supporting atomic operations on global memory for G80 cards with a Compute Unified Device Architecture (CUDA) computing capability of 1.1. They extended this with atomic operations on shared memory in compute capability 1.2 and on floating point values in compute capability 2.0. Ever since then, the architectural support has evolved with better design and greater efficiency. In the most recent GPU cards, with computing capability of 2.0 and above, atomics performance has greatly improved with the use of the last level cache. Atomic instruc-

tions are applied to avoid data race conditions and ensure the order of memory operations such that every read-modify-write operation to a shared memory location can happen without interference. Correspondingly, if two threads at the same time try to read-modify-write to the same memory address, they have to be serialized. This effect may lead to substantial performance degradation if there are a lot of atomic conflicts. Furthermore, atomic conflicts from threads that run simultaneously instead of running in parallel may prevent these threads from benefiting from context switches because of their cross-thread data dependence.

There are a few studies in the literature relevant to atomic collisions on GPUs or SIMD architecture. In [9], the authors try to add more efficient architecture support for atomic vector operations by using the similar vector gather/scatter hardware unit for irregular memory references. However, there have been very few software studies from this aspect. Most relevant software work is application specific and/or with certain underlying assumptions. In [2], the authors take advantage of GPU shared memory atomics to reduce global memory atomic conflicts in a bio-optics application. In [11], image histogramming is the major computing task and special characteristics of the image workload are used to help eliminate atomic collisions. Researchers also tried to use atomic operations to help improve the performance of parallel reduction on irregular workload in AMD GPUs [5]. However, the performance of the atomic operations themselves is not the focus. In summary, there is a lack of systematic investigation of GPU atomics from the aspect of software approaches. Many questions remain unanswered. For example, what are the differences in the influences of atomic collisions at different granularities, and the collisions from threads on one SIMD processor and the collisions from threads running on different SIMD processors? The atomic collisions vary from different runs of the same program with different inputs and the same run of the program with different phases. What type of collision distribution causes the most serious atomic conflicts? How can we quantify and characterize the atomic write collisions? What are the possible solutions to reduce atomic collisions dynamically? What are the trade-offs among different algorithms and different scenarios?

In this paper, we try to address these problems. We start by analyzing the behavior of different levels of atomic collisions experimentally. After a thorough analysis of write collision and its causes, we discover two key techniques essential for atomic collision reduction: reduction/scan based and reordering based. We further analyze the trade-offs of different algorithms. Moreover, we find that atomics can also greatly improve performance for traditional applications such as Sparse Matrix Vector Multiplication. In summary, the contributions of this paper are:

- We investigate the extensive usage of atomics in computation tasks for GPU applications for the first time. We analyze the trade-offs between programming designs involving atomics or not. The advantages include more active warps on the fly to hide the computation/memory latency, better load balancing, simplified program design, and finer-grained partitioning of the work load. The disadvantage is that if atomic operations are not used appropriately, the performance can degrade significantly.
- We identify that the cause of degraded atomic performance is the large amount of read-modify-write operations to the *same addresses* at the *same time* by *parallel threads*. We discover two key techniques for eliminating atomic collisions dynamically. The first one involves reducing the total number of atomic updates. The second one is to scatter conflicted atomic updates over different time periods to avoid serialization. Based on these two key techniques, we propose and analyze a set of efficient write-collision-elimination algorithms. With no prior knowledge of program data, our algorithms run efficiently and adaptively. Our algorithms are generic and can be applied to different programs regardless of their different characteristics and inputs.
- We analyze in depth the interaction of write collisions at different levels. We unify the solutions for removing atomic conflicts and provide insights for future exploration.

The rest of the paper is organized as follows. We provide motivation and discuss the performance bottleneck of atomic-intensive applications in Section 2. We describe the principles of reducing atomic collisions in Section 3. In Section 4, we discuss approaches to reducing atomics collisions solely within each SIMD processor. In Section 5, we consider simultaneous atomic collisions across different SIMD processors and propose solutions for them. We define and describe metrics to characterize the behavior of write collisions as well as an algorithm to estimate these metrics on GPUs in Section 6. We present evaluation results in Section 7, discuss related work in Section 8, and draw some conclusions in Section 9.

2. Motivation

The GPU is a highly parallel many-core architecture. It normally consists of hundreds or more cores on one chip. We will take NVIDIA GPUs as an example to explain the execution model and architecture of GPUs. A GPU chip features multiple Streaming Multiprocessors (SM). A Streaming Multiprocessor is a Single Instruction Multiple Data (SIMD) processor, in which a group of threads execute the same instruction on multiple data elements in parallel. In the NVIDIA CUDA programming model, a GPU kernel function is the part of code that executes on the GPU. It may invoke tens of thousands of threads to run simultaneously on a single GPU chip. These threads are organized into blocks, within which threads can synchronize and communicate through shared on-chip memory. A block runs on at most one Streaming Multiprocessor. A block of threads is further divided into warps. A warp is defined as a SIMD thread group whose component threads run in parallel on a SIMD processor. Multiple warps run simulta-

neously. Warps are also minimal scheduling units for context switches within every Streaming Multiprocessor. The size of a warp is typically 32 threads for NVIDIA GPUs.

There are several types of memory on a NVIDIA GPU card. We use the terminology defined for NVIDIA GPUs. Global memory, also called off-chip memory, is of larger size but slower than the on-chip memory types. The on-chip memory types include read-only constant memory and texture memory, which work as hardware-managed cache. The shared memory is on-chip memory that needs to be managed manually by the programmer. In the rest of the paper, we use the term "shared memory" to refer to the special type of software-managed on-chip memory on a GPU card and the term "global memory" to refer to the off-chip memory on a GPU card. The shared memory is orders of magnitudes faster than global memory. However, the shared memory is typically quite small, for example, 16KB per Streaming Multiprocessor on the Tesla 1060. Atomic memory operations are only available for global and shared memory, because only they can be both read and written. There are several important atomic functions for updates on shared data structures such as atomic add, subtract, logical and, logical or, etc.

Atomic collisions refer to scenarios where multiple parallel threads try to read-modify-write the *same data object* at *the same time*. With the guarantee of atomicity, these updates from different threads will be serialized. More specifically, threads within a warp that have conflicting memory write requests to the same object at one instruction have to be serialized. Threads within the same warp need to load, update, and store shared data structures as a unit, because that particular warp is blocked until all operands are ready. This is the implementation of NVIDIA GPU's global memory atomics (on Fermi-based architecture), which is based on specialized hardware for blendng. The GPU shared memory atomics are implemented differently. Not all threads have to have their operands ready before they can be context switched in. There is a loop that updates the available memory addresses by acquiring locks at each iteration and keeps updating until all threads are covered. A mask is used to mark which threads are done and which threads are not. In either case, write collisions may cause a slow down due to exclusive atomic memory accesses. In the former case, atomic collisions may cause thread divergence, because some threads in the same warp may be idle while others are not. The latter case will not cause thread divergence, but may reduce the total number of ready warps since a warp is ready if and only if no atomic addresses are locked.

There are two basic types of write collisions on GPUs. The first type of write collision exists only within warps, which means threads within the same warp may write to the same memory location in one instruction. We call these intra-warp collisions. The second type of write collision only exists across warps, which means different warps may have overlapping atomic memory destinations. We call these inter-warp

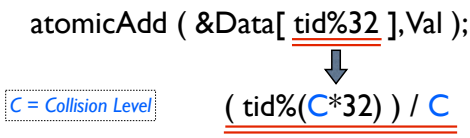


Figure 1: Motivation Example Code: tid is thread id. Redraw the graph if there is time, for every C, generate a row of boxes

collisions. The two types of write collisions can co-exist. Intra-warp collisions are fundamental since there are direct and immediate memory conflicts within a warp. Inter-warp collisions may also involve direct and immediate memory update conflicts if the different warps run on different SIMD processors at the same time. The inter-warp write collisions can be converted to intra-warp collisions by sorting the memory request addresses of all threads. The intra-warp write collisions can be converted to inter-warp collisions by spreading the same type of threads across different warps. Given that the memory access addresses are known in advance, we can first sort and create intra-warp collisions. If we can completely solve the intra-warp collision problem first, then we only need to handle the inter-warp collisions. However, it is difficult to solve the problem the other way around. Furthermore, the performance impact of inter-warp collisions is smaller than that of intra-warp collisions if we hold the number of collided memory objects constant. This is because inter-warp collisions mostly affect the latency-hiding efficiency obtained by context switching. Intra-warp collision directly affects the parallelism of a single vector operation as well as the context-switching efficiency. These results are further confirmed by micro-benchmark results, described in the next paragraph.

How harmful atomic collisions can be

We use two examples to show the influence of write collisions on GPU application performance. One is a micro-benchmark and the other is a real-world application. In Figure 1, we show the micro-benchmark code. Assume the warp size is 32. The first line indicates the data access pattern of different threads implied by their global thread IDs. There is only inter-warp collision and no intra-warp collision since the array access index for every thread within the same warp is different given the warp size 32. Next, we introduce a intra-warp write collision by changing the array reference indices from *(tid%32)* to *(tid%(C*32))/C* . We introduce a new parameter C to control the level of intra-warp collision. The parameter C represents the frequency of the same destination address occurring. For instance, if $C = 2$ and the original access pattern is {0, 1, 2, ... 31, 0, 1, ..., 31, ...}, then the new access pattern will be {0, 0, 1, 1, 2, 2, 3, 3, ..., 31, 31} and every address appears exactly twice in each warp. The set of memory accesses to the array D remains unchanged. We adjust the level C and measure the execution time of this micro-benchmark, which is shown as second row in Table 1. We also show slow-down of the program compared with the baseline, the original case when $C = 1$. The running time

C	1	2	4	8	16
Time (ms)	601	3815	12627	27892	60495
Slow Down	1	6.34	21	46	100

Table 1: Motivation Level Example

increases more than linearly with respect to the collision factor. The performance deteriorates significantly and it is up to 100 times slower when $C = 16$, regardless of the fact that the whole set of write requests remains the same. Furthermore, the overall number of collisions is the same. The results of the micro-benchmark confirm the conclusion we made in the last paragraph that intra-warp collisions can be more detrimental than inter-warp collisions. This is because when we keep the same number of collisions and represent them either as intra-warp collision form or inter-warp collision form, the inter-warp collision case always outperforms the intra-warp case. The intra-warp collision problem is more fundamental.

A More Realistic Example: Sparse Matrix Vector Multiplication In this paragraph, we show the influence of atomic collisions on a real application. We compare the performance of an atomics-involved computation kernel and non-atomics-involved computation kernel. For the atomics-involved case, we show two cases. One case is the native atomic write update version, in which we simply replace every write operation as an atomic write. The other case is more sophisticated, in which we added some scheduling and computation reduction operations. We use a kernel from CUSP, a library for sparse linear algebra and graph computations on CUDA. CUSP provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. CUSP libraries have been highly optimized and produce excellent benchmark performance results. They support the latest CUDA version and GPU device compute capability. We use the sparse matrix vector (SPMV) multiplication kernel using COO sparse matrix representation. The original CUSP sparse matrix vector multiplication does not have any native atomic operations. We implemented a native atomic write version which simply make updates to the result vector y using every element-element multiplication in $A * x$, where A is the sparse matrix and x is the vector. Next we implemented a more sophisticated approach based on a parallel reduction and scan algorithm, which will be described later in the paper. We used the benchmark matrices included in the CUSP library and we compared the results with the non-atomic CUSP implementation. In Fig. 2, the y-axis shows the speed up, which is the baseline CUSP implementation time divided by the two atomics-involved implementation times. The larger the speedup is, the better the performance is. If the speedup is below 1, then the atomics version runs slower than the baseline version, and otherwise it is faster. In Fig. 2, we can see that the native atomics implementation slowed down the program significantly in most cases, up to 5 times slower than the baseline. This is due to a large number of intra-warp write collisions in SPMV. For the optimized atomics implementation, we see a speedup of up to 1.7 times. This figure shows the large potential performance improvement with appropriate native atomic updates for GPU applications. Additionally, the length of the source code drops from 383 lines (including the kernel configuration) to 190 lines in the fastest atomics implementation.

3. Principles for Reducing Atomic Collisions

As discussed in Section 2, the root cause of degraded atomic performance is read-modify-write operations to the *same addresses* at the *same time* by many *parallel threads*. One solution is to gather the updates to the same memory addresses, and do preprocessing to perform some necessary computation on them before a collective write to the same memory address. An example is a reduction operation such as summing many data elements in one array. We refer to it as the *merging*-based approach in this paper. A completely opposite idea is instead of merging updates to the same destination before a collective write, spread them across different thread warps and across different time instants so that we can cause *different* destination addresses to be written to at *the same time*. We call the second approach the *splitting*-based approach. These two approaches have different complexities and different amount of overhead as well as different applicable scenarios. We describe the two types of approaches briefly in Section 3.1 and Section 3.2.

3.1. Merging-Based Method

In the *merging*-based method, we collect the operands and operators of atomic updates to the same address and perform a parallel reduction on them. The operands are data values and are thus independent of memory objects. Reduction here means performing the same operation on a set of input values and reducing them to one aggregated value. The operation can be addition, subtraction, logical operations or any other operator having the commutative property. The order of the operations should not matter, since if the order of these operations did matter, these operations would have to be serialized and no scheduling approach could improve the performance. In Figure 3, assuming a warp has 8 threads, we show both addresses and values for the atomic add operation. The number of colliding atomic writes is 7 for warp 1 since there are 7 add operations to memory address 0. After a parallel reduction on the first seven input values which turns out to be 16, we only need to perform one atomic add to increment the value at the memory location by 16. We can let any thread in this warp perform this add operation. In this example, we let the first thread in the warp perform the atomic update to memory location 0. For the last thread, which writes to memory location 1, we let it perform the atomic add with the first thread that performs the aggregated add operation. The total number of atomic updates can therefore be reduced from 8 to 2. The parallel reduction operation on GPUs has been extensively optimized. We use the most optimized parallel reduction [8] kernel and improve it by adding parallel scan operations so

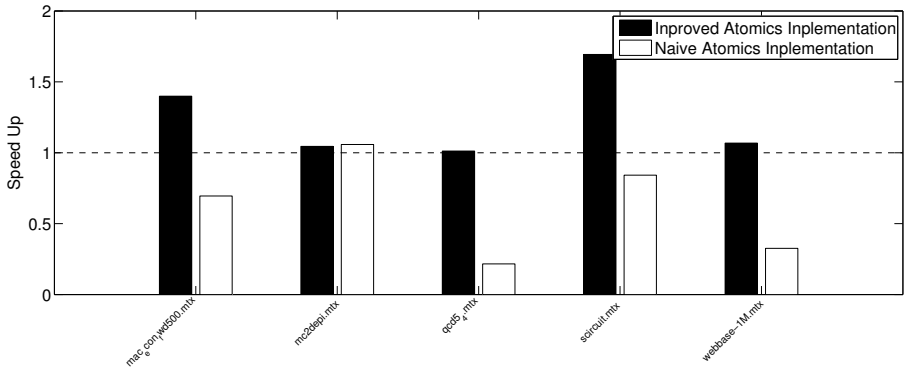


Figure 2: Using atomic operations on Sparse Matrix Vector Multiplication Library

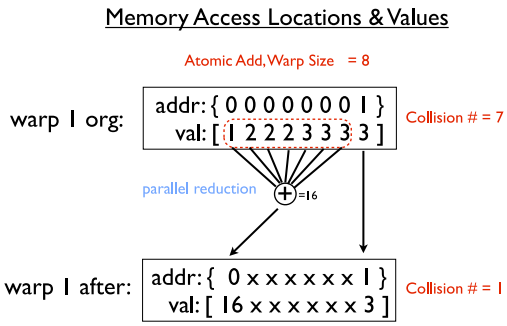


Figure 3: Merging Based Method Example

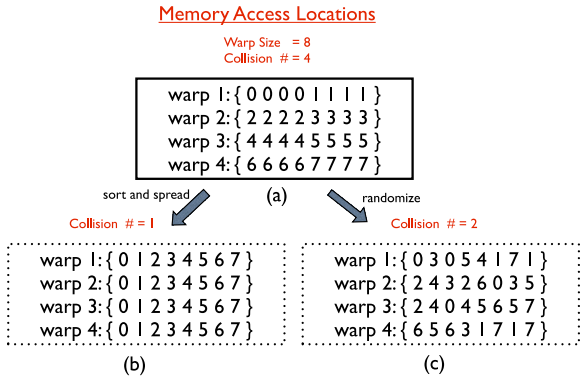


Figure 4: Scattering-Based Method Example

that non-aggregated and aggregated atomic adds can happen at the same time.

There are three main challenges to implementing the *merging* -based approach. The first one is, how can we determine the most frequent memory address for merging operation. Further, how can we see the data operands and destinations from other threads. The parallel reduction happens first, to generate aggregated memory update results. Only after that will different atomic memory updates happen. The second problem is the scenario where very few threads collide with each other. The atomic updates may need to be serialized because only aggregated updates happen at one time. In this case, using the original atomic functions may yield better performance. The third difficulty is, when atomic operations reside in conditional branches, the general parallel reduction algorithm does not work. We discuss these problems in Section 4.

3.2. Splitting-Based Method

We also describe the *splitting*-based method using a simple example in Figure 4. Assume the warp size is 8. In Figure 4 (a), we show the addresses of atomic memory operations for every thread from four warps. In the first warp, there are write collisions to memory locations 0 and 1, each repeated four times. In Figure 4 (a), the collision factor (we define collision factor as the maximum number of repeated memory accesses in a warp, see Section 6) is 4 for every warp. The basic idea of splitting is to reorganize the threads, whether splitting them into different warps or scattering different warps, so that the atomic update memory location changes and the number of collisions can be reduced for every individual warp. There are two different ways to reshuffle the threads into different warps, one is through data reorganization and the other is through thread relocation. This solution to the problem is analogous to a type of transformation that swaps jobs of every thread, discussed in detail in [17].

There are different ways to reshuffle the threads. In Fig. 4 (b), we first sort the threads based on their atomic memory destinations. Then we assign the threads into warps in a round-robin manner. Essentially every two adjacent threads are assigned to different warps. In Fig. 4 (b), after sort and reassignment, for every warp, there are no intra-warp collisions. The collision factor for every warp is 1, which is a 87.5% reduction from the original case. In Fig. 4 (c), another way to reshuffle threads is called *randomization*. Instead of sorting the threads based on their memory destinations, for each thread, we pick a new random thread location and assign it to this random location. In Figure 4 (c), after random shuffling, all four warps have a collision number of 2, which is 50% of the original case. The *randomization* also helps reduce

the collision number, though not as much as the sorting-based approach. However, the transformation overhead is smaller and no prior knowledge about the write collision addresses is needed. Note that both reshuffling approaches can be applied with different reshuffling scopes to strike a balance between the overhead and benefit. For example, the randomized reshuffling approach can happen either within or across thread blocks. If reshuffling happens within thread blocks, we can use shared memory to ensure coalesced global memory accesses.

4. Reducing Intra-Warp Collisions

In this Section, we focus on intra-warp collisions. Intra-warp collisions are fundamental. Here, we assume that atomic collisions only exist within warps, not across warps. The approaches we discuss in Section and are based on the basic prototypes described in Section 3. The *vote&reduce/scan* approach in Section tackles the problems discussed before for *merging*-based approaches.

4.1. Vote & Reduce/Scan Algorithm

The *merging*-based approach reduces the write collisions significantly but also adds more computation overhead. It takes up to five iterations ($\log(32)$) to complete a full parallel reduction/scan (if we want to update multiple write destinations at the same time) on a warp of size 32, while it takes the maximal number of iterations from the most frequent memory address destination in that warp. If the repeated number of writes to the same location is small, then reduction will serialize the thread execution, since we can only perform reduction for the same memory address at one time. However, when there is no write collision at all in a warp, the atomic instruction can be performed in parallel by all threads. The key challenge of applying the reduction is how to identify the most frequent atomic memory location. In the *Vote&Reduce* algorithm, we try to get the most popular atomic memory operation addresses and perform reduction on that if the benefits brought by reduction will offset the extra overhead.

In this algorithm (Algorithm 1), the variable *SampleTid* is the sampling thread ID for this warp, which is generated randomly. The *SampleId* is generated using a full cycle random number generator of cycle 256. For every warp, we generate a random number between 0 and 31 with 32 as the warp size. For every warp, the more frequent an atomic memory address is, the more likely the thread with that atomic address will be chosen. As the number of warps increases, the number of our trials increases, the more likely the probabilistic rule will hold. It is correct according to the *large number theorem*. The number of warps in a GPU kernel is typically large and thus our prediction strategy is quite efficient. We will discuss this more in Section 6.1.

After getting the *SampleTid*, we get the sample memory address for this particular thread, and do a poll from all the threads in one warp [1] to find out the threads that have the same destination address and the number of these threads. Therefore, we can determine which target atomic address to be used for parallel reduction. If the number of the threads sharing the same address is greater than a preset threshold (lines 5–12), we will perform reduction. For the rest of the threads that use different target atomic addresses, we let the them use the native atomic functions. If the count is smaller than or equal to the preset threshold, it means the amount of write collision is small, and a *merging*-based approach may only degrade the performance. In that case, we use the native atomic function. We choose the preset threshold of between 5–8 because for a warp of size 32, at most $\log(32) = 5$ iterations are needed. If the memory address collide for less than 5 times, then the sequential number of updates is also 5. We implement the *Vote&Reduce/Scan* version of atomic update for all the intrinsic atomic and commutative functions in CUDA using Algorithm 1. We show only the reduction implementation for a single destination address. We implemented and used a scan version which can perform reduction for multiple destination addresses at the same time which takes at most ceil$(\log(n))$ number of iterations.

Algorithm 1 The Vote & Reduce Atomic Update Algorithm

```
1:  Threshold = 5;
2:  SampleTid = Rand(Current Warp of Threads);
3:  SampleAddress = AtomicAddress[SampleTid];
4:  Freq=Vote(AtomicAddress[tid] == SampleAddress);
5:  if (Freq ≥ Threshold) then
6:     Reduction(SampleAddressValues, &Result)
7:     if (tid%WarpSize==0) then
8:        Update(SampleAddress, Result);
9:     end if
10:    if (AtomicAddress[tid] ≠ SampleAddress) then
11:       AtomicUpdate(AtomicAddress[tid], Val[tid]);
12:    end if
13: else
14:    AtomicUpdate(AtomicAddress[tid], Val[tid]);
15: end if
```

The *Vote&Reduce/Scan* approach can be implemented as compiler support with user specified annotations. Besides, with compiler support, we can further optimize by controlling the register usage which currently is a major problem, because making a function call (we currently implement runtime library) may add register pressure. The number of registers used determines the utilization of every SIMD processor. Note that, this approach requires extra shared memory space to save the sampled thread index and intermediate reduction values. The requirement is small compared to the total size of the shared memory available, for example, 4KB out 64KB of the latest Fermi architectures. We take compiler suppport for this as our future work.

[1] In NVIDIA CUDA, it is implemented with *__ballot() function.*

4.1.1. Atomics within Conditional Statements When atomic functions are called within a conditional statement, the reduction in *Vote & Reduce* needs to be reconsidered as well as the selection of the single thread that updates the reduction result. Because some threads may not execute the atomic function if the branch is not taken for that thread and therefore not the reduction function. And the single thread that finally updates the memory function may not always be the first thread ID in the warp. Our solution is to reassign every thread that is busy a new thread ID according to their order in the array of working threads. We also use shared memory to save the values that need to be updated atomically temporarily. Therefore, based on the new thread ID for every working thread, we perform the reduction and single thread atomic update correspondingly.

5. Reducing Inter-Warp Collisions

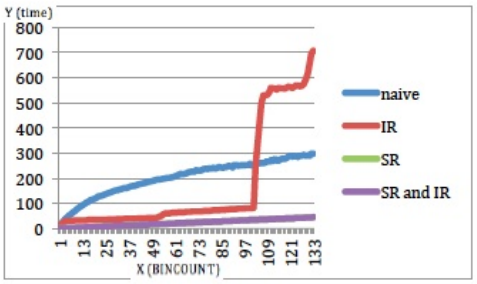
When inter-warp collisions exist, there are two scenarios. In the first case, the inter-warp collisions happen within thread blocks that run at the same Streaming Multiprocessor. The collisions will affect the efficiency of simultaneous multi-threading, since warps cannot be swapped in fast enough to hide the memory/computation latency due to the dependence on the data values of the collided memory locations. In the second case, the inter-warp collisions happen across thread blocks and also on different Streaming Multiprocessor. The second case is parallel atomic collision because different Streaming Multiprocessor may execute the atomic instruction and try to read-modify-update the memory location at the same time, which make the warps run sequentially and throttle the memory bandwidth. In this case, note that there are usually many warps which run simultaneously at the same Streaming Multiprocessor. The warp that is blocked due to the atomic collision will yield to other warps that are ready. Therefore, one may wonder if we can randomize the warps so that there are enough different warps for context switch and thus we can improvement performance. However, because now all the Streaming Multiprocessors share the same memory controller to global memory, the requests all go to the global memory and/or last level cache. We have observed that the NVIDIA GPUs use a random scheduling policy for choosing which warps to run from a set of blocks and choosing which block to run from a set of blocks. This can be modeled by a queuing network as illustrated in Fig. 5. A set of streaming multi-processors send requests a memory controller. If the destination address is the same, the requests will be queued in a buffer. We have observed from extensive experiments that the arrival process for memory requests to the shared global memory controller is a random process that can be modeled use poisson processes. Therefore, the queuing network can be modeled as a M/M/k queuing network. In order to keep a balanced flow for such a queuing network, which means the number of items in the queues stay in a steady state, the arrival rate of the requests should be less than or equal to the service rate of the queue. The service rate of a memory controller depends on the latency of a memory transaction and the arrival rate of memory requests depend on the number of streaming multi-processors and the program itself. For example, the latency of a global memory transaction is usually several hundred cycles, which is around 100 times of the processor instruction latency, the probability of colliding memory requests at the same time need to be less than 1/100. This has been confirmed in our experiments that updates a histogram with controlled number of bins and controlled frequency for every bin. In the left half of Fig. **??**, we show the performance when the number of bins increases. We find the IR version, which is basically a splitting based approach that scatters different warps as far as possible from each other, has a drastic turn point at the point of 100 bins using every possible approach. The reason is for this particular GPU architecture we used, the ratio between the processor rate and memory rate is around 100 times.

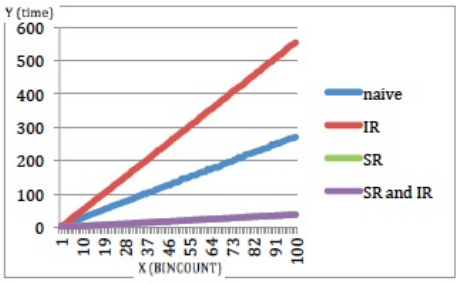
Theorem. *In a GPU kernel function with atomic operations, if the atomic operations phase of the function is the bottleneck compared to all other phases of the function, and if all atomic operations are for the same memory with one memory controller with no private coherent cache [2], then the performance of the GPU function is invariant to the way the warps are placed on different SIMD multiprocessors.*

Proof. The above theorem can be proved using queuing theory. In Fig. 5, we model a GPU kernel function execution as a queuing model. The incoming requests on the right are from large amount of threads simultaneously running the system. Every node in this network represents a server. For the atomic operations, first we read the data from memory and then we perform computations on SIMD processors (represened as circles in the figure), modify the data and write back to memory. Then this request from a paticular thread is served and will move on to the next phases of the program. According to queuing theory, if one node in the queuing network is the bottleneck which means it run slower than all the other nodes, the queue for this node will grow exponentially and the throughput of the whole network is determined by the throughput of this particular node. Therefore, if the memory atomics stage in this queuing network is the bottleneck, the throughput will be all the same no matter which thread warps reside in which SIMD processor. The total amount of time will be the same as the processing time of all atomic operation requests. The theorem is then proved. □

According to the Theorem and Proof above, for the first case, even if the inter-warp collisions happen within every SIMD processor, as long as the atomic requests for global memory, the warp scheduling does not work. If, for the first case, the inter-warp collisions occur due to shared memory atomic requests, the randomization approach may make differences. It

[2] In NVIDIA GPUs, if there is private L1 cache, it is not cache coherent and therefore is normally bypassed by atomic operations





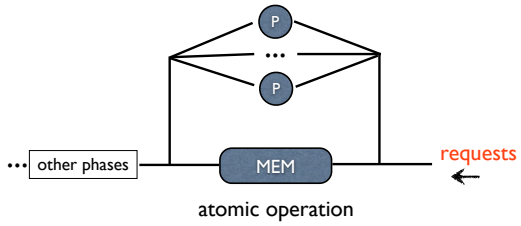


Figure 5: Queuing Model for Inter-warp Collisions: "P" represents SIMD processors, "MEM" represents memory controller or last level cache

is because shared memory is private to every SIMD processor and thus rescheduling warps to different thread blocks may help scatter the similar warps to different SIMD processors and reduce atomic collisions wihin single SIMD processor. Another insight we can obtain from the above Theorem is that when global memory atomic operations are not bottlenecks of a GPU function, then the theorem does not hold. However, in this case, we don't need to optimize the performance of atomic operations. We have also confirmed the validity of the theorem by experimenting with micro-benchmarks, which show insensitive performance behavior when we change the placement of thread warps.

To alleviate the performance degredation of inter-warp atomic collisions, there are several solutions. First, it is the merging based approach for intra-warp level, which can reduce the pressure for global memory atomics collisions. Secondly, the thread randomization approach discussed in Section 4 can still help since it may reduce the influence of thread serialization within every SIMD processor. We will show the effectiveness of these two approaches in Section 7.

In summary, the intra-warp merging approach and thread randomization approach are major effective techniques for atomic collisions elimination. It works for both inter-warp collisions or intra-warp collisions or the combination of both. These two approaches can further be combined if further knowledge of the program data can be known in advance. For example, we can sort the memory addresses and put threads that access same memory location in the same warp, and further reschedule warps across different warps to help the shared memory atomics. The techniques we proposed here are fundamental software approaches that can be extended and we we leave this as our future work.

6. Finding the Best Algorithm

6.1. Characterization of Atomic Write Collisions on GPUs

Metrics: We define a few metrics for write-collision detection. The first one is Collision Number (CN). It is defined as the maximum number of repeated memory accesses in one warp at one atomic instruction. The second metric is Collision Factor (CF), which is defined as the average of collision number for all the warps.

Assume N = total number of warps, then CF is:

$$CF = \sum_{k=1}^{N} CN_k/N$$

.

For collision number, if the atomic operations is implemented by spinlock, then it represents the largest number of iterations a loop takes for this particular warp. We choose the largest because of the lockstep behavior of SIMD processor. The other threads that have fewer iterations have to wait for the threads that take longer time because one instruction is issued at one time for all threads. If the atomic operation is truly implemented by a hardware atomic instruction, then this represents the total number of times this memory location is read and updated by this warp. The larger the collision number is, the longer time it takes for this warp to complete the atomic function. The collision factor is the average of collision number, which represents the extent of write collisions among all the threads.

The next metric is called Multi-Collision Number (MCN). It is defined as the average of the number of repeated memory accesses for the top Mmost frequent memory locations in one warp.

$$MCN(M) = \sum_{k=1}^{M} Freq(MemTop_k)/M$$

.

Correspondingly, the Multi-Collision Factor (MCF), is the average of multi-collision number over all warps.

$$MCF(M) = \sum_{k=1}^{N} MCN_k(M)/N$$

.

The multi-collision number and multi-collision factor are used to extract the distribution information of write collision for all the warps over all threads. The single collision number is for the most frequently accessed memory address for atomic updates. If the single maximum collision number is very large (the maximum is the warp size = constant), then we know the distribution of frequent write-collision mem addresses. However, if the single maximum collision number is medium, it is hard to tell how frequent the other write-collision memory destinations are. With the multi-collision number, we can easily find out whether there are multiple frequent memory addresses or not. This fine grained information is important for the selection of best atomic collision reduction algorithm. We will discuss this more in Section 6.2.

Algorithm 2 Probabilistics-Based MCF Estimation GPU Kernel

```
1:  Count=0;
2:  SampledPoints={};
3:  CandidateThreadPool = {Current Warp of Threads};
4:  while Count < M do
5:     SampleThreadId = Random(CandidateThreadPool);
6:     if (AtomicAddress[SampleThreadId] not in Sampled-
       Points) then
7:        SampleAddress = AtomicAddress[SampleThreadId];
8:        Freq = Vote(AtomicAddres[tid] == SampleAddress);
9:        if (tid%WarpSize == 0) then
10:          MCN[tid] += Freq;
11:       end if
12:       SampledPoints.add(MemSamplingPoint);
13:       CandidateThreadPool.remove(SampleThreadId&Other);
14:    end if
15:    Count++;
16: end while
17: MCN[tid] = MCN[tid]/Count;
18: GlobalSync;
19: MCF = Avg(MCN);
```

To estimate these metrics, we developed a probability-based GPU algorithm which is highly parallel and accurate. The basic idea of this algorithm is to randomly select a thread in the warp. Then obtain the atomic memory destination for this thread, and vote within the same warp to get the frequency of this particular address. Since the thread ID is randomly selected, the frequent atomic addresses among all the threads may be selected with large probability. Meanwhile, there is a large number of warps running simultaneously which makes sure the probabilistic estimation close to its expected value. This algorithm is not only useful for atomic collision characterization but also for dynamic decision making about whether the optimization should be applied and how the optimization should be applied, because it is very fast, running nearly at full bandwidth. The algorithm to estimate MCF(M) is listed as Algorithm 2. In Line 5, the algorithm randomly picks the sampling point and the popular atomic memory locations get picked with larger probability. Once a sampling point is selected, we will start voting on this memory address and save the frequency into $MCN(tid)$. We keep sampling and voting until we finished sampling for M most frequent memory locations. In the end, in lines 17-19, the Multi-Collision Numbers will be averaged over the number of times sampling and the total number of warps. The algorithm runs on GPGPU and utilizes the statistical property of large amount of GPU threads, which is highly parallel. We will show the effectiveness of the write collision detector in Section 7.

6.2. Choosing the Best Algorithm

Both aggregation based and scattering based approaches have their advantages and limitations. With different distribution of the memory atomic addresses from all threads, one approach may out-perform the other. For example, if there is large amount of repeated atomic access to the same memory location across all the threads and there are very few distinct memory locations to be written to, the *splitting method* does not work well. Because no matter how we reshuffle the data layout, there will still be write collisions within warps. On the other hand, for the distributions that have many repeated memory access locations but medium amount of repeats for each location, the reduction approach may incur too much overhead and slow down the atomic memory accesses. For example, assume there are two warps, which have these access patterns: {1 1 3 3 4 4 7 7 } and {2 2 5 5 6 6 0 0 }. The reduction based approach can perform reduction on one value at once, so there will be at least four iterations of reduction for each warp while using the native atomic operations only involves two iterations of memory updates. However, if we use reshuffling, we can make both warps like this {1 2 3 5 4 6 7 0} and there will be no intra-warp write collisions. In this case, the reshuffling approach works better than the reduction approach. Finally, if the distribution is totally random such that there is no write collision within warps, there is no need to apply either type of approach.

Recall that in Section 6.1, we described different metrics to model the amount of write collision and the type of write collision distribution. Using Collision Number(CN) and Collision Factor (CF), we set a threshold such that if CF is large enough, then it means in every warp, one memory destination location dominates and we should use *aggregation* based approach. Using the Multi-Collision Number(MCN) and Muli-Collision Factor(MCF), if CF is small but MCF is consistently the same number as CF, then we can apply *scattering* based approach. We further evaluate the usage of different metrics and their

implications in Section 7.

7. Evaluation

We run experiments on the NVIDIA GTX 480 card with CUDA computing capability 4.0. The GTX 480 is NVIDIA Fermi [12] architecture. It has in total 64KB reconfigurable L1 cache and shared memory per Streaming Multiprocessor, and 768KB shared last level L2 cache. The host machine is equipped with 2.27 GHz Intel Xeon E5520 chipset and linux 2.6.37. We select benchmarks that involve non-trivial atomic operations from a realistic domain of applications such as data mining and scientific simulations. There are respectively: Image Histogramming, GPU-Mcml and Fluidanimate. The timing results are derived from more than five repeated runs for every configuration. For every benchmark, we evaluate both merging based approach and splitting based approach. For the randomization experiments in the category of splitting based approaches, we pre-generate the random sequence on CPUs which can be easily reused after for different GPU kernel functions. We implemented a runtime library including the set of atomic-collision elimination and characterization algorithms.

7.1. IMAGE HISTOGRAMMING

Many image processing applications use the histogramming algorithm, which fills a set of bins according to the frequency of occurrence of pixel values taken from an input image. We used the image histogramming benchmark developed and optimized in [11]. The authors of this benchmark provided warp-private histogramming (which use warp-private shared memory histograms to improve the performance) and thread-private histogramming implementation. The thread-private histogramming implementation does not involve atomics. We extended this benchmark by implementing block-private histogramming and no-private histogramming approaches in order to test our software techniques with different types of atomic write collisions. We used real images with different distributions.

We present the results in Table 2. We choose 9 images with different amount of write collisions. For every image, we estimate the collision factor and then obtain the real collision factor to evaluate the prediction accuracy. CF represents Collision Factor (CF), and MCF represents Multi-Collision Factor. For MCF, we choose the top 2 most frequent memory addresses for every warp. We can see from the table that the prediction accuracy is satisfactory. There is slight error which may be due to missing some warps. Note that, the real collision factor is no smaller than the estimated collision factor because in the worst. The statistical sampling would miss maximum frequency items and thus the overall collision factor is smaller. In the other half of Table 2, we show the performance of different types of approaches. "SpeedUp(W)" represents the code version that uses warp private histogram. "SpeedUp(B)" represents the code version with block private histogram. "SpeedUp(N)" represents the code version without using any private histogram to reduce atomics collisions. We can see that for most cases, either *Merging* or *Splitting* approach out-performs the original version significantly. For the case that no private histogram is used, the speed up brought by different techniques are tremendous probably because of the baseline is low and also because the two approaches we have proposed are robust in different scenarios. Note that it seems the *Merging* version outperforms the *Splitting* version for most of the time. The reason is that the overhead of reshuffling data for *Splitting* is much larger compared to the benefits it can obtain. In fact, in several cases, without taking the overhead into account, the *Splitting* performs significantly better than *Merging*. This is something we would like to further explore in the future about how to reduce the reshuffling overhead.

7.2. Sparse Matrix Vector Multiplication

We have presented the results for Sparse Matrix Vector Multiplication in previous Sections. We present more results in this section in Table **?? ??**. When the matrix is very sparse, the atomic-naive has best performance when it is moderate sparse, the merging based approach has large performance improvements sometimes is better than non-atomic but splitting based approach is worse than even atomic-naive When it is dense, merging based approach has little performance improvements and is worse than non-atomic, but splitting based approach has a lot of improvements compared to atomic-naive approach we used is mainly merging based.

7.3. GPU-MCML

GPU-MCML [2] is a highly optimized Monte Carlo (MC) code package for simulating light transport in multilayered turbid media. It is the GPU version of the widely cited MC code package in biophotonics called MCML which is a sequential C code. Atomic operations are to the absorption array and have been known as performance bottleneck if not handled properly. We present the results for GPU-MCML in Table 5. The GPU-MCML application uses shared memory to cache data in order to reduce the atomic conflicts in global memory, which carries a much larger latency. We can improve the performance of GPU-MCML slightly using the *Merging* approach. The result of randomization approach is not very satisfactory due to the fact that the original regular memory access patterns are affected, which brings more performance degradation. There are many iterations in the loop, in each iteration the distribution of atomic addresses may be changed. We sampled the first iteration and provide the collision factor number for that iteration.

7.4. CUDA-CUTS

Graph Cuts is a powerful and popular optimization tool for energies defined over an MRF and has found applications in image segmentation, stereo vision, image restoration etc. The maxflow/mincut algorithm to compute graph cuts is compu-

Image Input	CF		MCF		SpeedUp(W)		SpeedUp(B)		SpeedUp(N)	
	Estm.	Real.	Estm.	Real.	Merge	Split	Merge	Split	Merge	Split
Auroa	11	13	14	16	1.64	1.14	2.22	0.62	3.4	0.97
ContemporyArt	31	31	31	31	2.37	1.26	3.45	1.40	16.5	1.4
ComplementBear	8	12	12	16	0.98	1.15	0.86	0.80	1.05	1.26
SystemDesign	24	26	26	28	1.57	1.10	2.25	1.02	1.83	9.97
Star02207	31	31	31	31	5.24	0.85	9.9	0.90	18.7	0.97
Star03968	9	11	11	13	1.65	0.98	1.32	0.91	1.83	0.942
Star4080	20	21	21	22	2.10	0.818	4.85	0.84	6.28	0.95
Star4094	20	21	23	24	2.57	0.76	4.67	0.83	6.0	0.92
Star4103	18	19	19	21	3.12	0.83	4.64	0.88	7.3	0.97

Table 2: Experiment Results of Write Collision Detector on Image Histogramming

BMK Group	attributes			original				
	MAX	MIN	AVERAGE	non-atomic	atomic-naive	Merge	Split.	Merge and Split.
consph	9.719	0.120	8.655	1.298	9.694	1.741	14.16	1.743
mc2depi	0.076	0.038	0.076	0.663	0.614	0.619	0.837	0.732
mac_econ	2.131	0.048	0.299	0.608	0.792	0.352	0.997	0.341
dense2	10000	10000	10000	0.883	20.62	1.107	8.688	1.113
pdb1HYS	56.02	4.943	32.76	0.984	7.026	1.241	11.05	1.248
rail4248	514.2	0.009	24.10	3.103	48.23	3.285	24.56	3.253
scircuit	20.64	0.059	0.328	0.482	0.540	0.282	0.634	0.258
webbase	47.00	0.010	0.031	0.971	2.362	0.968	3.095	0.731

Table 3: Benchmark Groups

tationally expensive. *CUDA-CUTS* [**?**] is an implementation of the push-relabel algorithm for graph cuts on the GPU. The GPU version performs much faster than its counter part CPU version, and the atomics-involved implementation performs better than the non-atomics-involved implementation. We can see that neither merging or spliting approach performed well because of the very few write collisions due to the conditional writes (collisions are spread into different time instants).

8. Related Work

There have been few studies on the atomic collisions on GPGPUs. Atomic operations are known to be very expensive, especially on GPUs where thousands of threads can easily run simultaneously and exacerbate the situation. Programmers have been tryting to avoid using atomic functions in applications that need to deal with large amount of shared memory structure such as parallel sorting [15] and reduction operations.

A most relevant work is the hardware extension for efficient atomic vector support [9], where the authors study the SIMD vector atomics in Chip Multi-processors (CMP) with a SIMD width of 4. However, in the GPUs, the SIMD width is far more than 4. In Fermi-architectures, it is 32 which exposes more challenges for atomic operation optimizations. In NVIDIA GPUs, the speed of atomic operations have been improved on Fermi-based [12] architectures thanks to a combination of more atomic units in hardware and the addition of the L2 cache. However, it is still important to optimize atomic accesses to both GPU shared memory and global memory. Most relevant work for optimization in this aspect are application specific. One example is GPU-MCML[2], which is a highly optimized Monte Carlo (MC) code package for simulating light transport developed on NVIDIA Fermi GPUs. In this paper, the authors used shared memory to help improve the global memory atomic performance. Another example is real time hashing, in which the algorithm have multiple hash functions and round robin on collision until they insert or fail. Besides, there is work on improving histogramming algorithms by reducing atomic collisions in various ways such as using private shared memory and applying workload specific predictions [13] [16] [14] [11].

To the best of our knowledge, this work is the first one that systematically studies the influence of atomic collisions on GPGPUs, and explores a set of software solutions. One of the approaches we used is statistical voting based approach to quickly identify the frequent atomic accesses. There is related voting based approach for Optimal Binary Prefix Sum [3], but not for atomic collisions. We also propose to use reduction in the *merging* based approach. Reduction has been extensively studied in different types of GPUs including NVIDIA [8] and AMD GPUs. In [5], the authors try to improve the performance of reduction using atomic memory udpates. The *splitting* based approach is relevant to the job swapping idea used in control divergence elimination for GPU kernels [17] while the goals are different. In control divergence removal, the

BMK Group	scatter				
	non-atomic	atomic-naive	SR	IR	SR and IR
consph	1.300	9.689	1.741	14.19	1.745
mc2depi	0.662	0.615	0.619	0.837	0.733
dense2	0.883	20.62	1.107	8.674	1.113
mac_econ	0.609	0.793	0.352	0.996	0.341
pdb1HYS	0.984	7.026	1.241	11.05	1.249
rail4248	3.103	48.22	3.285	36.02	3.239
scircuit	0.482	0.540	0.282	0.634	0.258
webbase	0.970	2.363	0.968	2.834	0.730

Table 4: Benchmark Groups

Method	SpeedUp	Org. CF	Alg. Choice
Merging	1.1	5	Yes
Splitting	0.83	5	No

Table 5: Experiment Results of *Gpumcml*

Method	SpeedUp	Org. CF	Alg. Choice
Merging	0.95	1	No
Splitting	0.76	1	No

Table 6: Experiment Results of *CUDA-CUTS*

goal is to make threads convergen within the same warp while in atomic collision removal, the goal is to make threads as different as possible.

9. Conclusion

In this paper, we systematically studied the influence and solution of atomic collision problems on GPGPUs. We proposed novel techniques to estimate atomic collisions and extract their statistical distribution information. We explored different types of atomic collision reduction algorithms based on two fundamental techniques: the *merging*-based approach and *splitting*-based approach. Both types of approaches are efficient in different scenarios. We tackled the difficulties in implementing each type of algorithm, such as the dynamic detection of frequently colliding memory addresses. We designed statistical approaches combined with a thread voting scheme to characterize and provide guidelines for selecting and/or applying the best atomic collision removal algorithm. As far as we are concerned, this is the first work that studied the atomic collision problem in a general software framework instead of under software-specific or hardware-specific assumptions. This is also the first time an efficient parallel statistical voting approach is introduced to solve the atomics performance problem on GPUs.

References

[1] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Building an efficient hash table on the GPU," in *GPU Computing Gems*, W. W. Hwu, Ed. Morgan Kaufmann, Aug. 2011, vol. 2, ch. 1.

[2] E. Alerstam, W. C. Y. Lo, T. D. Han, J. Rose, S. Andersson-Engels, and L. Lilge, "Next-generation acceleration and code optimization for light transport in turbid media using GPUs," *Biomedical Optics Express*, vol. 1, no. 2, pp. 658–675, 2010.

[3] D. W. Anthony Skjellum and P. Bangalore, "Ballot counting for optimal binary prefix sum," in *GPU Technology Conference*, ser. Poster, 2010.

[4] B. Coutinho, D. Sampaio, F. Pereira, and W. Meira Jr, "Divergence analysis and optimizations," 2011.

[5] B. Dhanasekaran and N. Rubin, "A new method for gpu based irregular reductions and its application to k-means clustering," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/1964179.1964182

[6] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, "Frequent itemset mining on graphics processors," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, ser. DaMoN '09. New York, NY, USA: ACM, 2009, pp. 34–42. [Online]. Available: http://doi.acm.org/10.1145/1565694.1565702

[7] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 407–420. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.12

[8] M. Harris, "Optimizing parallel reduction in cuda," 2007.

[9] S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen, "Atomic vector operations on chip multiprocessors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 441–452. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.38

[10] K. S. Matthew Sinclair, Henry Duwe, "Porting cmp benchmarks to gpus," *Technical Report - University of Wisconsin Madison*, 2011.

[11] C. Nugteren, G.-J. van den Braak, H. Corporaal, and B. Mesman, "High performance predictable histogramming on gpus: exploring and evaluating algorithm trade-offs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 1:1–1:8. [Online]. Available: http://doi.acm.org/10.1145/1964179.1964181

[12] NVIDIA, "Nvidia's next generation cuda compute architecture: Fermi," 2010.

[13] ——, "Cuda c programming guide 4.0," 2011.

[14] V. Podlozhnyuk, "Histogram calculation in cuda," in *Technical Report*. NVIDIA, 2007.

[15] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1586640.1587667

[16] R. Shams and R. A. Kennedy, "Efficient histogram algorithms for NVIDIA CUDA compatible devices," in *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, Gold Coast, Australia, Dec. 2007, pp. 418–422.

[17] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '11. New York, NY, USA: ACM, 2011, pp. 369–380. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950408